Pre-Silicon Validation of 32-Bit Indigenous Processor for Space Applications

S. J. Anjana, K. Padmakumar, Joji Daniel, L. S. Syamlal, Ganta Nagendra Mourya Teja, K. Ranjani, R. Paramasivam, and M. Narayanan Namboodiripad

1 Introduction

The processor on-board the launch vehicle performs navigation, guidance, control and stage sequencing functions. Since the launch vehicle missions are designed with zero defect as the target, it is highly imperative that the processor design is fully validated in the ground before the chip is fabricated. Insufficient validation had resulted in the infamous Pentium division bug, necessitating a major recall of the Pentium computers [1]. Computer architecture refers to those attributes of a system visible to a programmer and has a direct impact on the logical execution of a program [2]. For a microprocessor, instruction set architecture (ISA) can be taken as its specification. Validation shall ensure that the correct result is always obtained for a sequence of valid instructions, and it shall also detect the potential errors and design deficiencies.

For the custom-designed 32-bit processor, hereafter referred as PROC32, the most prevalent technique in the industry, simulation-based validation is used, with additional test cases at each stage of design cycle. Another standard technique of formal verification, wherein specification can be proven to satisfy a given set of properties, was not used for this fairly simple design, where the time to delivery of product was a constraint. Elaborate test programs developed by the designers and standard validation suites for IEEE754 floating-point data were used for ISA validation. Since PROC32 design is non-pipelined and without cache, simultaneous interactions between different design elements and timing uncertainties do not affect its functioning. Thus, systematically crafted test cases can achieve complete

S. J. Anjana (✉) K. Padmakumar · J. Daniel · L. S. Syamlal · G. N. M. Teja · K. Ranjani
R. Paramasivam · M. Narayanan Namboodiripad
Vikram Sarabhai Space Centre (VSSC), Thiruvananthapuram, India
e-mail: sj_anjana@vssc.gov.in; syamlal_ls@vssc.gov.in

© Springer Nature Switzerland AG 2021

M. Palesi et al. (eds.), *Second International Conference on Networks and Advances in Computational Technologies*, Transactions on Computational Science and Computational Intelligence, https://doi.org/10.1007/978-3-030-49500-8_17

design validation of PROC32. The methodology presented in this paper involves testing the microprocessor design using a combination of VHDL simulator and FPGA prototype and verifying the outputs against those of instruction simulator and standard test suites. The paper discusses the test case design strategy, test platform and validations done for characterising the reliability of PROC32. The paper concludes with the summary of performance metrics.

2 Literature Survey

In a typical engineering project on system on chip (SoC), verification effort is significantly high, consuming more than 57% [3]. The key goal of system validation is to ensure that its behaviour is correct and consistent. It encompasses functional and performance verification. The designers need to start detecting the bugs early in development, continue testing till the silicon wafer is fabricated and should be able to reuse the vectors once the chip is out from the foundry. For the low-level bugs in a processor design to be brought out, the internal state of its microarchitecture is to be assessed by applying stimuli like assembly programs to cover the entire design space.

3 Processor Overview

The indigenisation elements in PROC32 include the processor architecture, hardware design, chip fabrication and system software tool suite, as depicted in Fig. 1. ISA has been defined primarily to meet the functional and safety requirements of flight software. In-house design ensures that the chips can be re-fabricated based on demand. The indigenous device manufacturing leaves no scope for bug intrusion like Trojans. The in-house system software tool suite supports customisation of user requirements.

PROC32 is designed around von Neumann architecture, ensuring deterministic performance for real-time applications. To avoid catastrophes, fraction arithmetic, which works on a closed set of values $(-1, +1)$, is adopted and overflows are indicated through hardware arithmetic errors (AEs). PROC32 is an accumulator-based machine with microprogrammed control logic, simple addressing modes and 152 fixed-length instructions operating on fixed-point, floating-point and integer data

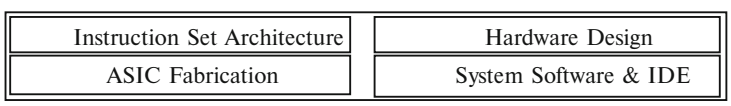

Fig. 1 Indigenous 32-bit processor ecosystem

types. The address space supports large code and data size for embedded real-time applications. Hardwired floating-point unit (FPU) also implements trigonometric instructions.

For faster memory access, on-chip RAM is provided for software, application data, operating system (OS) data and stack. SECDED (single bit memory error correction and double-bit error detection) protection is provided with enable/disable options. Programmable memory wait states facilitate interfacing with devices of different access times. Four programmable counters enable real-time scheduling, and a watchdog timer helps to detect and recover from any deadly situation like hanging of processor. About 256 software interrupts and 12 hardware interrupts are supported. Two on-chip indigenous MIL-STD-1553 protocol controllers serve as input–output communication interfaces.

4 Instruction Set Architecture

The programming model includes 26 registers, each 32-bit long. Two accumulator registers are used for 32-bit and 64-bit operations. Seventeen index registers enable efficient data access across multiple segments. Loop Counter facilitates iterative execution of instructions. Program counter keeps track of the sequence of instructions to be executed. Stack Pointer manages the stack. Status Register setting controls interrupts and write protection of system memory area, besides reflecting the hardware arithmetic errors. Three instruction formats permit operands in memory (Type-I), constant operands (Type-II) and operands in registers (Type-III). Two addressing modes are provided for operands with respect to program counter or index register using 20-bit signed offset field in the opcode. Little endian convention is used for data access. Generic data formats like 16-/32-bit integer (signed/unsigned), real number (IEEE754) and fraction arithmetic are supported. Two's complement arithmetic is used for fixed-point instructions. A 64-bit float has a precision of 15 decimal digits and a range of $1.79 \times 10^{+308}$ and 32-bit float has a precision of 6 decimal digits and a range of $3.40 \times 10^{+38}$.

Instruction set has been evolved to meet the real-world data processing requirements. Special instructions like multiply and accumulate (MAC) are intended for digital signal processing applications like filters. Dedicated instructions are provided for microdiagnostics (CPU/memory tests), interrupt handling and to load/read timers. Pre-scale instructions select the clock frequency of operation of the real-time counters. Instructions are defined to configure the wait states for memory-mapped peripherals and to transfer data between 16-bit peripherals and 32-bit processor. Test and set instruction enables semaphore operation by controlling access of resource by multiple processes in a multi-tasking system. Stack operations support modular software structure. Bit manipulation instructions (Set/Reset/Read) aid ON/OFF control in applications like programmable logic controllers. Special instructions process blocks of data, like array copy, initialisation and checksum computation for integrity checks.

5 Test Methodology

The processor testing is a challenging area and requires in-depth understanding of the CPU design. In microprogrammed control unit, microcode translates machine instructions into sequences of detailed circuit-level operations. Each microinstruction of PROC32 is a 55-bit long word, functionally divided into 20 fields. It generates ALU control signals and address of next microinstruction. CPU takes multiple clock cycles to execute a single machine instruction, one for each microprogram step. A dedicated FPU performs floating-point, trigonometric and data type conversion and MAC operations.

For PROC32, the software development tool suite is designed and developed in-house. Ada cross compiler translates the source programs in Ada into equivalent assembly code. Assembler converts assembly language programs to re-locatable machine code. Linker links the object code with the system libraries to generate executable code with physical addresses assigned. This is executed in the instruction simulators and target hardware, using the setup in Fig. 2. The vectors generated from the test programs can drive the RTL model in VHDL simulator running on 3.4 GHz Intel core i7 machine with 8 GB of memory. The internal state of processor, with registers, memory and system bus, is fully observable and controllable in this test scenario.

In simulation-based validation, each test is first executed in PROC32 simulator and then run on the design under test (DUT), which is the Register Transfer Logic (RTL) implementation of the microprocessor. Graphical user interface (GUI) and command-line versions of microcode and instruction simulators enable testing in PC by modelling the instruction set, registers, memory and execution flow of instructions. These generate execution trace and clock statistics. The single-stepping feature of the simulator is very useful in the initial design phase. The inputs and outputs are consolidated in a table. It is ensured that the hardware AE bits in Status Register are set in case of arithmetic overflows and software AE word bits are set in case of abnormal inputs to the software library routines. The deviations of actual outputs from references are traced using VHDL simulator, and the defects rectified after thorough analysis.

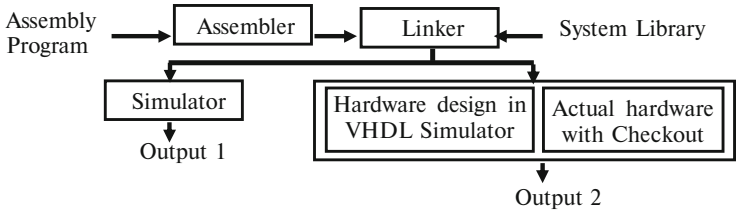

Fig. 2 ISA validation setup

6 Test Case Design Strategy

Test cases were judiciously chosen to satisfy the objectives of ensuring maximal functionality coverage and fault coverage as explained below. Statement coverage is a measure of the ability of test programs to access every executable statement in code. Code coverage of 99 % was achieved for PROC32 in a total of 3×10^8 simulation cycles. The major bugs that necessitated code changes are discussed in Sect. 7.

6.1 Functionality Coverage

Test programs are designed to exercise all the instructions for various input combinations in the different addressing modes so as to ensure the correctness of implementation of the ISA. Assembly-level processor self-test programs written for 152 instructions achieve 100% functional coverage. Corner case data values and inputs outside the functional domain are fed to the instructions, and the response is characterised. Table 1 summarises the statistics of test cases for different types of instructions.

The instructions are tested using inputs derived from equivalence partitioning, a standard technique for revealing errors with less number of test cases, reducing the test time. Boundary value testing is carried out to verify the behaviour of instructions for nominal and boundary values. Floating-point instructions are tested with additional inputs like denormal, infinity and NaN (Not a Number), and outputs are checked for overflows and underflows. It is verified that denormal inputs

Table 1 Instructions versus test cases

Instructions	Test cases
Floating-point 32-/64-bit Add/Sub/Mul/Div/Neg/Abs/Compare (12) + trigonometric functions Sin, Cos, Arctan, Sqrt, Exp and Log (12) + MAC (2) + data type conversion (10)—36 nos	6000 each—216,000 nos
Fixed-point and unsigned arithmetic and logic 16-/32-bit Add/Sub/Mul/Div/Neg/And/Or/Xor—28 nos	20 each—560 nos
32-bit integer Mul/Div/MAC—3 nos	25 each—75 nos
Logical shift 16-/32-bit (max 15/31 shifts)—4 nos	25 each—100 nos
Arithmetic shift 16-/32-bit (max 15/31 shifts)—6 nos	25 each—150 nos
Control transfer (branch/call/return/compare)—18 nos	15 each—270 nos
Load/store 16-, 32-, 64-bit (8), peripheral interfaces (2), bulk data process (3), bit operations (3), semaphore (1)—17 nos	5 each—85 nos
Register copy, initialisations, sign extension—20 nos	2 each—40 nos
Control instructions (interrupt, timer, memory protect)—13 nos	15 each—200 nos
Stack (push/pop/save/restore)—4 nos	2 each—8 nos
Microdiagnostics (CPU/memory tests)—3 nos	1 each—3 nos

Table 2 Representative test matrix

Instructions	Typical inputs
16-bit signed arithmetic	−32768, −32767, −1, 0, 1, 32766, 32767
32-bit signed arithmetic	−2147483648, −2147483647, −32768, −1, 0, 1, 32767, 2147483646, 2147483647
16-bit unsigned	0, 1, 32767, 32768, 65534, 65535
32-bit unsigned	0, 1, 32767, 65535, 4294967294, 4294967295
16-bit shift (logical and arithmetic)	0_h, 1_h, $7FFF_h$, 8000_h, $FFFF_h$, 5555_h, $AAAA_h$; No. of shifts—0, 1, 2, 3, 13, 14, 15
32-bit shift (logical and arithmetic)	0_h, 1_h, $7FFFFFF_h$, 80000000_h, $FFFFFFF_h$, 55555555_h, $AAAAAAAA_h$; No. of shifts—0, 1, 2, 15, 29, 30, 31
32-/64-bit floating point	+0.0, −0.0, +min, −min, +nominal, −nominal, +max, −max, +Infinity, −Infinity, +sNaN, −sNaN, +qNaN, −qNaN, denormal, −denormal
Trigonometric (32- and 64-bit floating point)	π/2, π/3,π/4, π/6, −π/2, −π/3, −π/4, −π/6, other special cases in floating-point representation
Data type conversions	Values in the range of source and destination data types

are treated as ±0.0 and underflows result in ±0.0. The precision and rounding scheme are ensured as per design. Data type conversion and MAC instructions are tested with out-of-range values, besides others. Floating-point MAC instructions are verified through matrix multiplication outputs too. Standard FPU verification test suite [4] was downloaded from the internet, and 117,470 test cases were executed. Programs are designed to test different sequences of valid instructions too. The register and memory contents are verified before and after execution of each instruction, to ensure the behaviour as per specifications. The functional outputs from VHDL simulator and Status Register contents, indicating arithmetic errors if any, are verified against the reference outputs, along with the execution clock cycles. Table 2 features the representative test inputs.

The library routines, which invoke appropriate processor instructions after input pre-processing, are also tested in the target. Trigonometric functions are implemented using CORDIC (coordinate rotation digital computer) algorithms in the processor design. For verifying these, test programs are written in Ada and Assembly and the linker-generated binaries are executed in the VHDL simulator. Reference outputs are generated using C++ programs that invoke the corresponding standard mathematical libraries available in PC compiler and also log the floating-point status flags. The CORDIC functions implemented in microcode simulator serve as another reference.

6.2 Fault Coverage

Test vectors are generated for the fault coverage of memory. Programs are written to access all memory locations (load/store) with patterns intended to detect 'stuck-at' faults, facilitating toggle coverage (the percentage of bits that toggle at least once from 0 to 1 and at least once from 1 to 0 during the simulation). The exercise was repeated to ensure coverage of all registers too. These vectors were also delivered to the foundry where the processor chip is fabricated, for clearing functionality. Tests using Automatic Test Pattern Generation (ATPG) are used subsequently by the chip manufacturer to expose any electrical and manufacturing defects within the chip.

7 Validations Carried Out

7.1 CPU Testing

CPU test instruction checks the ALU microcode operations (3 Arithmetic and 5 Logical operations), flags (Overflow, Carry, Zero and Negative) and external registers (Status Register, Memory Address Register, Memory Data Register and Micro-Index Register). Self-testing of 152 instructions is carried out as detailed in the previous sections.

7.2 Error-Handling Capabilities

Arithmetic overflows, floating-point operations with NaN and divide by 0 conditions result in hardware AE bit in the Status Register, leading to the consolidated exception bit (EX), triggering EX interrupt. OS can then take appropriate error-handling actions in the Interrupt Service Routine (ISR). Status Register also indicates underflow, inexact representation and denormal inputs for floating-point data, as shown in Fig. 3.

7.3 Interrupts and Exceptions

PROC32 supports 12 hardware interrupts, of which only 9 are connected. These include illegal opcode interrupt, SECDED-related double-bit, single bit and check bit memory interrupts, two timer interrupts and two 1553B protocol controller errors. The interrupted program context is saved in system memory area by a

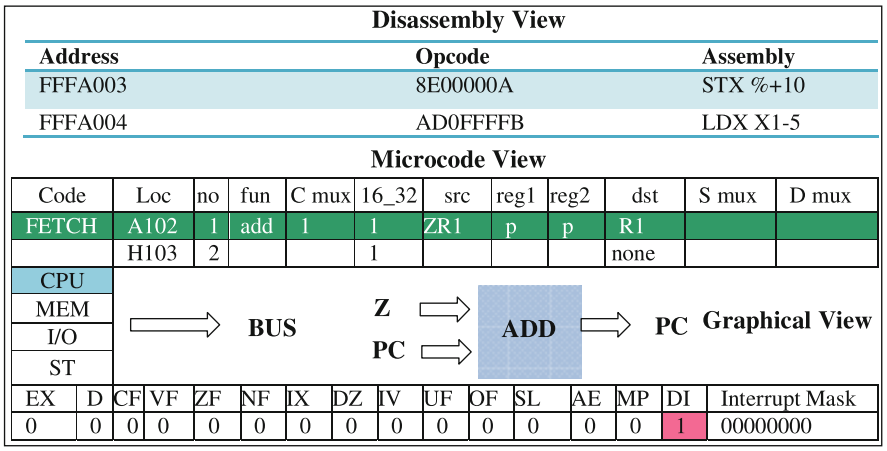

Fig. 3 Microcode simulator snapshot

hardware process transparent to the application task executing at that instant. Test programs are written to inject errors to trigger the hardware interrupts. About 256 software interrupts are tested by invoking instructions. Exception conditions are generated by selecting inputs that would lead to arithmetic saturation. It is verified that PROC32 handles the abnormal situations gracefully, giving control to the software to initiate recovery actions.

7.4 Memory and Peripherals

Assembly instruction is available to check the integrity of boot code in PROM. CRC (cyclic redundancy check) checksum and XOR checksum checks are implemented for PROM and tested. RAM health is tested by means of March pattern and Walk pattern tests. SECDED error detection/correction unit based on modified Hamming code for internal RAM is tested by injecting error after disabling SECDED and verifying after enabling it. Memory wait states are configured for different blocks through instructions and memory access times measured. For testing input/output interfaces, all patterns ranging from 0000_h to $FFFF_h$ are sent from checkout system to PROC32, received back through 1Mbits/sec 1553B link and compared. Real-time counters are verified by timer test programs executing for a fixed duration, the counter frequency being derived from processor operating clock frequency using pre-scale options.

7.5 Flight Software Evaluations

The confidence on the processor increases when it is proven in the actual domain for which it is primarily designed. Thus, launch vehicle flight software, with a representative mix of fixed and floating-point instructions, is thus a natural candidate for assessing the performance of the processor. Flight software is developed in Ada Language for Flight Software Application (ALFA) and tested using PC-based Ada compiler. Real-Time Executive (REX) software, comprising of boot code, static task scheduler and application interfaces, is developed in Assembly language and ALFA. Ada modules are cross-compiled to generate assembly code files. System Integrator utility integrates these modules with the REX software to generate a single assembly language file. The assembled flight code is linked to generate the binaries and downloaded to the memory of a Xilinx FPGA prototyping board, which is programmed with the VHDL design of PROC32. Figure 4 represents the proto-board for tests.

Simulated input profiles (SIPs) were fed through a checkout system, and the flight software execution outputs were compared with the reference outputs of a trajectory simulation program running in PC. The software for different launch vehicles like PSLV, GSLV and GSLV MkIII were tested, and the outputs for the nominal, $+3\sigma$ and -3σ cases were matching with the reference outputs of trajectory simulation software.

Fig. 4 FPGA prototyping board for tests

7.6 Processor Timing Performance

The timing performance has improved by around 75%, compared with the 16-bit on-board computer. This can be attributed to increase in word length of the processor, hardware FPU, more index registers, powerful instructions and higher clock speed of PROC32. The number of assembly instructions per Ada statement has also reduced considerably, due to the improvements in the architecture and associated compiler optimisations, resulting in a reduction in flight software code size by around 40%.

8 Observations and Corrections

Some typical observations during validation and corrections implemented are listed. The results indicate that the test suite can detect design errors and data-specific bugs.

8.1 Floating-Point Instructions

Observation Related to Accuracy for Trigonometric Instructions

The accuracy requirements for sine, cosine, arctangent and logarithmic functions are 6 and 15 decimal digits for single and double-precision floating-point numbers, respectively. But, the mantissa of the output was not exactly matching with the expected values for small input angles. [(Actual − Expected)/Expected] exceeded 10^{-6} and 10^{-15} for small values of 32- and 64-bit floating-point numbers, respectively.

Analysis and Correction

The accuracy loss is due to inappropriate setting of the number of CORDIC iterations in the algorithms. Also, the values of the constants used in the algorithms were not large enough. The constants were appropriately modified for accuracy.

The constants used to represent iterations in the CORDIC algorithms were appropriately modified for the single and double-precision floating-point functions of sine, cosine, arctangent and logarithm to meet the accuracy requirements.

Observation in Arctangent Instructions

During corner case tests for 32-bit and 64-bit arctangent instructions accepting two arguments with an operational domain of [0..1], errors in Table 3 were observed:

Table 3 Observation in arctangent instructions

Instruction	Inputs	Reference outputs	Hardware output
32-bit	00800001_h ($1.17e^{-038}$), 1.0	00800001_h	40000000_h (2.0); No error bit set
64-bit	8010000000000001_h ($-2.22e^{-308}$), 1.0	80100000 00000001_h	$C000000000000000_h$ (-2.0); No error bit set

Table 4 Observation in 64-bit floating-point negate/absolute instructions

Inputs	Reference outputs	Hardware output
8000000000000001_h ($-4.94e^{-324}$)	8000000000000001_h; Denormal bit set	0000000000000001_h (denormal); No error bit set
8040000000000001_h ($-1.78e^{-307}$)	0040000000000001_h	8040000000000001_h; Denormal bit and EX bit set

Analysis and Correction

Arctangent function with two arguments involves division operation (y/x). Root cause for the observation is absence of small angle approximation. The instruction is modified to accept a single argument, with small angle approximations. Division of inputs is performed by software before invoking arctangent function.

Observation in 64-Bit Negation/Absolute Instructions

While testing instructions on target, denormal bit was not set in the Status Register for some denormal inputs. The bit was wrongly set and instruction not executed for certain normal numbers, with exception bit set, as shown in Table 4.

Analysis and Correction

For double-precision numbers, single precision negate/absolute operation implementation was reused. Denormal check was implemented improperly, without considering the differences in representation of exponent between single precision (8-bit exponent) and double-precision (11-bit exponent) numbers. Negation and absolute operations are implemented by manipulating only the sign bit, and no other operation, like check for denormal numbers, is performed.

Observation Related to Denormal Numbers

The FPU design conforms to FTZ (flush to zero), which flushes denormal results to zero when the application is in the gradual underflow mode. Instructions involving denormal numbers as inputs are not processed by the FPU. Output of an operation

Table 5 Observation in denormal floating-point numbers

Operation and inputs	Reference outputs	Hardware output
Denormal + normal	Sum	Denormal; 'D' and 'EX' bits set
Normal + denormal	Sum	Normal; 'D' and 'EX' bits set
Atan2 (1.0, 00000001_h)	Π/2	Π/4; 'D' and 'EX' bits set

with denormal number input is same as the first operand. 'D' bit was set leading to 'EX' bit setting in Status Register, which is further used by software. Also, correct control flow was not ensured after comparison operation involving at least one denormal value. Typical example is given in Table 5.

Analysis and Correction

Denormal inputs are not processed by the instructions. The first input is returned as the result, with the exception also raised. The control flow takes the path corresponding to 'Less' in a floating-point comparison. DAZ (denormal numbers-are-zero) concept is introduced, which treats denormal inputs to floating-point instructions as zero. Also, 'D' bit set in such cases is prevented from setting 'EX' bit.

Observation Related to NaN

FPU returns qNaN as outputs for instructions involving qNaNs and sNaNs. Invalid ('IV') bit is set leading to 'EX' bit setting in Status Register, which is used by software for further error handling. Also, correct control flow is not ensured for comparison operations with NaN.

Analysis and Correction

As per IEEE754 standard, comparison between NaN and any floating-point value x yields a result NaN $\neq$ x. The FPU causes the control flow to take the path corresponding to 'Less' in a floating-point comparison. The design is modified to set Control Invalid bit ('CIV') bit in Status Register, in addition to 'IV' and 'EX' bits, so that software can take appropriate recovery action.

Observation for Truncation Instruction

FPU was returning infinity as output for truncation instruction, used to convert a double-precision floating point to single precision, for certain inputs. For example,

for $3.40e^{+038}$ ($47EFFDFFFDFFFFFF_h$), hardware returned infinity ($7F800000_h$) with 'OF' and 'EX' bits set, instead of the expected value of $7F7FF000_h$.

Analysis and Correction

The result is wrong when unbiased exponent of result is 127, and mantissa needs to be rounded. The output saturation happens irrespective of the mantissa bits. Exponent of result is incremented only when mantissa overflow happens.

8.2 Non-Floating-Point Instructions

During signed and unsigned instruction tests on target, certain deviations were observed. After corrections, instructions were re-tested and outputs were as expected.

Observation 1 and Correction

In Shift Left Arithmetic 16-bit instruction, MS 16 bits were not made 0 when number of shifts is 0. MS 16 bits of the result are forced to 0 in case of no shifts.

Observation 2 and Correction

In 32-bit unsigned Multiply and Divide instructions, Negative bit in Status Register was set when MS bit of 32-bit result is 1. Unsigned Multiply and Divide instructions are corrected so as not to affect the Negative (N) flag in Status Register.

9 Conclusion

The single-core 32-bit processor design follows simple fetch-and-execute von Neumann architecture, which restricts simultaneous access of an instruction and data. The single instruction stream and single data stream processing in PROC32 facilitates exhaustive evaluations of the design with a reasonable effort. This is unlike other state-of-the-art multi-core pipelined processors, with multiple instruction streams and multiple data streams, where the verification requires focus on the instruction-level parallelism. Systematic verification and validation activities were carried out for demonstrating the adequacy and suitability of PROC32 for meeting the workloads of flight software computations. The excellent performance in the

benchmarking flight applications establishes the correctness of the design decisions made. The enhanced throughput and reduction in code footprint have resulted from the architectural improvements, coupled with compiler optimisations, in PROC32, over its predecessor.

The systematic testing undergone on the hardware description language (HDL) design enables post-silicon validations to be conducted in a reduced timeframe. The test vectors already developed can be reused for assessing the functionality of the chip from the foundry. The computer system realised using the processor core fabricated as Multi-Chip Module (MCM)/Application Specific Integrated Circuit (ASIC) will undergo extensive validations in various test beds and also several qualification tests at different environmental levels. Once inducted in various launch vehicle missions of ISRO, this processor will find a place in similar safety-critical applications of other R&D organisations.

References

1. Halfhill, T.R.: The truth behind the Pentium bug. Byte, New York (1995)
2. Stallings, W.: Computer organization and architecture: designing for performance. Pearson, New York (2000)
3. Hrishikesh, M.S., Rajagopalan, M., Sriram, S., Mantri, R.: System validation at ARM enabling our partners to build better systems (2016)
4. Berkeley TestFloat. http://www.jhauser.us/arithmetic/TestFloat.html. Accessed 20 June 2019